Programmation Distribuée

1. Introduction à la Programmation Distribuée

La programmation distribuée consiste à développer des applications qui s'exécutent sur plusieurs machines interconnectées via un réseau. Ces systèmes permettent de résoudre des problèmes complexes en répartissant les tâches sur plusieurs nœuds de calcul et jouent un rôle essentiel dans la gestion de volumes massifs de données (le **Big Data**).

Système Distribué

Un système distribué est une collection d'ordinateurs autonomes qui apparaissent à l'utilisateur comme un système unique et cohérent. Les machines communiquent via un réseau pour coordonner leurs actions, offrant ainsi une grande puissance de calcul, une haute disponibilité et une évolutivité.

2. Motivations et Applications

Pourquoi développer des Systèmes Distribués ?

- **Aspects économiques** : Meilleur rapport performance/prix grâce à l'utilisation de plusieurs machines peu coûteuses.
- **Adaptation structurelle** : Alignement avec la structure géographique ou fonctionnelle des applications.
- **Intégration** : Nécessité d'intégrer des applications existantes dans un système unifié.
- **Communication et partage d'information** : Facilitation de la communication et du partage de ressources entre différentes parties.
- **Haute disponibilité** : Création de systèmes résilients capables de fonctionner même en cas de défaillance d'un nœud.
- **Évolutivité** : Capacité à évoluer et à s'adapter aux besoins croissants en données et en calcul.

Applications Distribuées

Une application distribuée s'exécute sur plusieurs machines et implique le traitement coopératif de données réparties. Ces applications ne reposent pas sur un seul nœud, mais distribuent les tâches pour améliorer performances et fiabilité.

Exemples de Services Distribués

- **Services de nommage (ex: DNS)** : Gestion des noms et adresses des ressources.
- **Services de communication** : Échanges de messages entre nœuds.
- **Services de fichiers distribués** : Gestion de fichiers stockés sur plusieurs machines.
- **Services d'exécution distribués** : Répartition des tâches de calcul.
- **Services de transaction distribués** : Gestion des transactions sur des bases de données réparties.

3. Programmation Classique vs. Distribuée

Programmation Classique

Les services sont appelés localement via des fonctions ou méthodes. Les programmes s'exécutent sur une seule machine en utilisant les ressources locales (mémoire, CPU, etc.).

Programmation Distribuée

Les services sont appelés à distance (souvent via une architecture client-serveur), ce qui masque la complexité des communications réseaux. Les programmes s'exécutent sur plusieurs machines en utilisant des ressources distribuées.

4. Problématiques et Défis

La programmation distribuée soulève plusieurs problématiques :

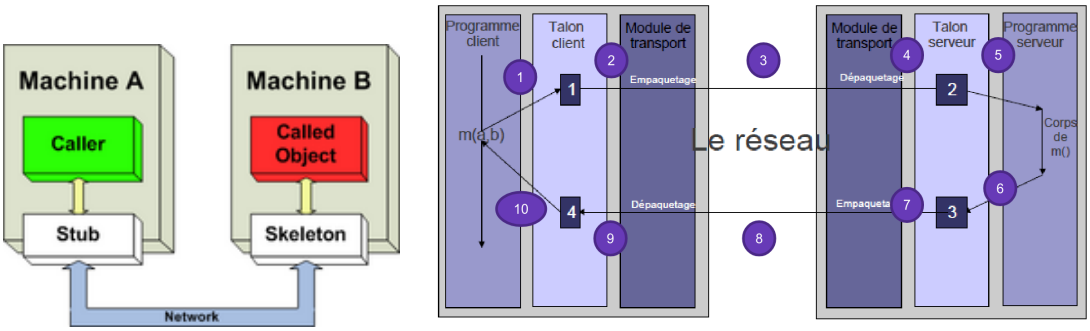
- **Interopérabilité des langages** : Conversion des représentations de types de données entre langages différents.
- **Passage de paramètres** : Gestion complexe des paramètres (pointeurs, références) pour qu'ils soient valides sur toutes les machines.
- **Localisation des services** : Recherche et accès efficace aux services distants.
- **Sécurité** : Garantie de l'authentification, de la confidentialité et de l'intégrité des échanges.
- **Performance** : Les appels distants sont beaucoup plus lents que les appels locaux, nécessitant des optimisations.

Hugo Tondenier

5. Technologies et Mécanismes de Communication

Appel de Procédure Distante (RPC)

Le RPC permet d'exécuter une procédure située sur une machine distante comme si elle était locale. Il masque la complexité du réseau grâce à des **stubs** (talons) qui gèrent la sérialisation et la désérialisation des données.



Fonctionnement

1. Le client appelle une procédure locale (stub client).
2. Le stub client sérialise/marshalise les paramètres et envoie la requête au serveur.
3. Le stub serveur reçoit la requête, désérialise les paramètres et exécute la procédure.
4. Le résultat est renvoyé au stub client, qui le désérialise avant de le transmettre au programme.

Avantages et Inconvénients

- **Avantages :**
  - Simplifie l'appel de services distants (syntaxiquement identique à un appel local).
  - Masque les détails du réseau (sockets, conversions de données, etc.) dans le stub.
- **Inconvénients / Problèmes :**
  - Gestion des défaillances (réseau, serveur, client).
  - Complexité dans le traitement de paramètres complexes (pointeurs, références).
  - Problèmes liés à la sémantique des appels (par exemple, "au moins une fois" ou "au plus une fois").
  - temps de réponse

Attention

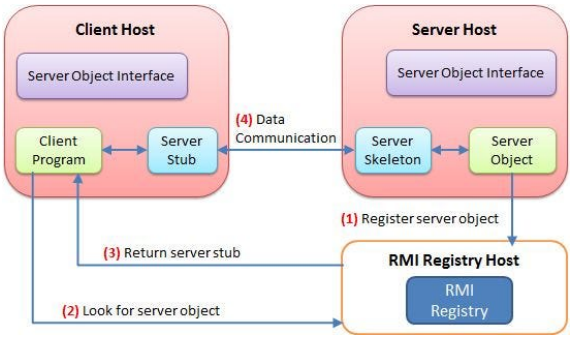
- Quand on fait du RPC, on passe les paramètres par valeur.
- Les pointeurs et passages de paramètre par références n'ont aucun sens hors de la machine client. $\Longrightarrow$ Ne jamais en utiliser avec RPC.
- On ne fait presque jamais de procédures car le serveur doit retourner quelque chose au client à la fin de l'appel.
- Une conversion est nécéssaire si les types ne sont pas encodés pareil sur le client et le serveur.

Remote Method Invocation (RMI)

RMI est une implémentation de RPC pour Java, permettant d'invoquer des méthodes sur des objets distants comme s'ils étaient locaux. Il utilise des stubs et des squelettes pour gérer la communication entre machines virtuelles Java (JVM).

Fonctionnement

- Le client invoque une méthode sur un objet distant via un stub.
- Le stub sérialise les paramètres et envoie la requête au serveur.
- Le serveur reçoit la requête, désérialise les paramètres et exécute la méthode.
- Le résultat est renvoyé au client, qui le désérialise avant de le retourner à l'application.



Technologie d'Échange de Messages (Message Exchange)

Contrairement au RPC, cette technologie repose sur une infrastructure de type **Message-Oriented Middleware** (MOM). Les messages transitent via une file d'attente (centralisée ou décentralisée), avec :

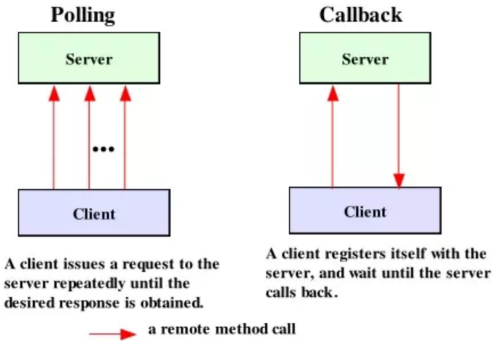
- Transmission asynchrone permettant une plus grande flexibilité.
- Utilisation de protocoles standards comme JMS (Java Message Service).
- Mécanismes de persistance qui améliorent la tolérance aux défaillances.

Avantages et Utilisations

- Communication déconnectée entre applications.
- Adapté aux systèmes où les délais ne sont pas critiques (ex. messagerie, systèmes financiers).

Callbacks

Les callbacks permettent au serveur de notifier le client lorsqu'un événement se produit. Le client s'enregistre auprès du serveur, qui le prévient dès que l'événement survient.



Utilisation et Exemple

- Souvent implémentés via un modèle de publication/abonnement.
- Par exemple, un serveur peut notifier un client dès qu'une tâche est terminée, permettant ainsi une communication asynchrone.

5.1 Localisation des Serveur

Pour exécuter un appel distant, il est essentiel de localiser le serveur et le processus associé. Deux approches principales existent :

- **Base de données centralisée** :
  - Un serveur enregistre les services disponibles dans une base centralisée.
  - Les clients consultent cette base pour localiser le serveur capable de fournir le service.
- **Serveur de nommage** :
  - Chaque serveur maintient une base locale des services qu'il offre.
  - Les clients doivent connaître à l'avance quel serveur interroger.

Solution alternative : Utilisation de serveurs décentralisés pour répartir la charge et améliorer la résilience.

6. Gestion des Échecs et Fiabilité

Échec du Client et du Serveur

- **Échec du Client** : Le serveur peut détecter un échec (par exemple via des timeouts) et gérer les appels orphelins. Le client peut relancer la requête avec une sémantique "au moins une fois".
- **Échec du Serveur** : Le client doit être capable de relancer la requête. Le serveur doit gérer les états intermédiaires afin d'éviter toute incohérence.

Fonctions Idempotentes

Les fonctions idempotentes peuvent être exécutées plusieurs fois sans provoquer d'effets secondaires, par exemple la lecture de données statiques.

Sémantique "At Most Once"

La sémantique "au plus une fois" garantit qu'une procédure distante est exécutée une seule et unique fois, même en cas de retransmission par le client. Pour cela, le serveur utilise des identifiants uniques pour chaque requête afin de détecter les doublons. Si une requête est reçue plusieurs fois, le serveur ignore les duplicatas et ne réexécute pas. Utile pour les fonctions non idempotentes

7. Sécurité dans les Systèmes Distribués

- **Authentification** : Vérification de l'identité des clients et serveurs.
- **Confidentialité** : Protection des données durant leur transmission pour éviter toute interception non autorisée.
- **Intégrité** : Garantie que les données ne sont pas altérées durant leur transmission.

8. Performance et Optimisation

Comparaison : RPC vs. Appel Local

Les appels via RPC sont quelques milliers de fois plus lents que des appels locaux. Cela nécessite des optimisations spécifiques dans les systèmes distribués.

Optimisation

L'utilisation de caches et d'autres techniques d'optimisation peut réduire le nombre d'appels distants et améliorer ainsi les performances globales du système.